

Controlling program extraction in Elementary Linear Logic

Marc Lasson

École Normale Supérieure de Lyon, France

marc.lasson@ens-lyon.org

We present an adaptation, based on program extraction in elementary linear logic, of Krivine & Leivant's system FA_2 . This system allows to write higher-order equations in order to specify the computational content of extracted programs. The user can then prove a generic formula, using these equations as axioms, whose proof can be extracted into programs that normalize in elementary time and satisfy the specifications. Finally, we show that every elementary recursive functions can be implemented in this system.

Introduction

Elementary linear logic is a variant of linear logic introduced by Jean-Yves Girard in an appendix of [3] that characterizes, through the Curry-Howard correspondence, the class of elementary recursive functions. There are two usual ways to program in such a light logic: by using it as a type system of a λ -calculus or by extracting programs from proofs in a sequent calculus (see [2] for instance).

The former is used for propositional fragments of Elementary Affine Logic in [6] and of Light Affine Logic in [1]. However, when the programmer provides a λ -term which is not typable, he has no clue to find a suitable term implementing the same function. In the later approach, the programmer must keep in mind the underlying computational behaviour of his function during the proof and check later, by external arguments, that the extracted λ -term implements the desired function.

In this paper, we describe a system in which we try to make the second approach a bit more practical. Firstly because our system is endowed with a kind of proof irrelevance: all proofs of the same formula are extracted to extensionally equivalent terms; and then because the program automatically satisfy the given specification used as axioms during the proof.

FA_2 is an intuitionistic second-order logic whose formulas are built upon first-order terms, predicate variables, arrows and two kind of quantifiers, one on first-order variables and the other on predicate variables. Jean-Louis Krivine described in [4] a methodology to use this system for programming with proofs. In this system, the induction principle for integers may be expressed by

$$\forall X, (\forall y, Xy \Rightarrow X(sy)) \Rightarrow X0 \Rightarrow Xx.$$

This formula is written Nx and it is used to represent integers. The programmer then gives some specifications of a function. For instance for the addition, he may give:

$$\begin{aligned} plus(0, y) &= y \\ plus(s(x), y) &= s(plus(x, y)). \end{aligned}$$

Now, if he finds a proof of

$$\forall xy, Nx \Rightarrow Ny \Rightarrow N(plus(x, y))$$

in which he is allowed to rewrite formulas with the specifications, then it is proved that the λ -term extracted from this proof using standard techniques is a program satisfying the specifications.

We have adapted the system FA_2 of Leivant and Krivine following two directions:

- We replace the grammar of first-order terms by the whole λ -calculus. We can then extract higher-order functions instead of purely arithmetical functions. We have shown in [5] that the resulting system can be described as a pure type system (PTS). We have also built an extensional model, and re-adapted realizability tools for it. Here we only present the material needed for elementary programming and we refer the reader to [5] for more details.
- We ensure complexity bounds by making its logic elementary.

In the next section, we introduce the grammar for our formulas and describe how we interpret them. In section 2, we present our proof system and how we can program with it. In the last section, we prove that we characterize the class of elementary recursive functions. We bring our system back to the usual Elementary Affine Logic in order to have the correctness. Finally we give two proofs of the completeness: one by using the completeness of *Elementary Affine Logic* (henceforth EAL) and the other by invoking, like in [2], Kalmar's characterization of elementary functions. We present the second proof as an illustration of how to program in our system. Indeed, it will give the programmer a direct way to code elementary functions without having to encode them in EAL.

1 Types, First-Order Terms and Formulas

We assume for the rest of this document that we have at our disposal three disjoint sets of infinitely many variables:

- the set V_0 of so-called *type variables* whose elements are denoted with letters from the beginning of the Greek alphabet and some variations around them (ie. $\alpha, \beta, \alpha_1, \alpha_2, \dots$),
- the set V_1 of *first-order variables* whose elements are denoted with letters from the end of the Latin alphabet (ie. x, y, z, x_1, x_2, \dots),
- the set V_2 of *second-order variables* whose elements are denoted with uppercase letters from the end of the Latin alphabet (ie. X, Y, Z, X_1, X_2, \dots).

We also assume that we have an injection of second-order variables into type variables and write α_X the image of a variable X by this injection. This will be useful later when we will send formulas onto system \mathcal{F} types by a forgetful projection.

Definition 1. The following grammars define the terms of the system:

1. Types are system \mathcal{F} types:

$$\tau, \sigma, \dots := \alpha \mid \forall \alpha, \tau \mid \sigma \rightarrow \tau$$

2. First-order terms are Church-style λ -calculus terms:

$$s, t, \dots := x \mid (st) \mid (t \tau) \mid \lambda x : \tau. t \mid \Lambda \alpha. t$$

3. Finally, second-order formulas are given by the following grammar:

$$P, Q, \dots := X t_1 t_2 \dots t_n \mid P \multimap Q \mid \forall X : [\tau_1, \dots, \tau_n], P \mid \forall x : \tau, P \mid \forall \alpha, P \mid !P$$

Theses grammars describe terms that will be used in this paper, λ, Λ and the three different \forall behave as binders like in usual calculi. We always consider terms up to α -equivalence and we do not bother with capture problems. We also admit we have six notions of substitution which we assume to be well-behaved with regard to the α -equivalence (all these notions are more seriously defined in [5]):

1. the substitution $\tau[\sigma/\alpha]$ of a type variable α by a type σ in a type τ ,

2. the substitution $t[\tau/\alpha]$ of a type variable α by a type τ in a first-order term t ,
3. the substitution $t[s/x]$ of a first-order variable x by a first-order term s in a first-order term t ,
4. the substitution $P[\tau/\alpha]$ of a type variable α by a type τ in a formula P ,
5. the substitution $P[t/x]$ of a first-order variable x by a first-order term t in a formula P ,
6. the substitution $P[Q/X x_1 \dots x_n]$ of a second-order variable X by a formula Q with parameters x_1, \dots, x_n in a formula P .

The last one is not very usual (the notation comes from [4]): it replaces occurrences of the form $X t_1 \dots t_n$ by the formula $Q[t_1/x_1] \dots [t_n/x_n]$ and it is not defined if P contains occurrences of X of the form $X t_1 \dots t_k$ with $k \neq n$. The simple type system we are going to define will guarantee us that such occurrences cannot appear in a well-typed formula.

And since we can build redexes in terms (of the form $((\lambda x : \tau. t_1) t_2)$ and $((\Lambda \alpha. t) \tau)$) we have a natural notion of β -reduction for first-order terms which we can extend to formulas (we write $t_1 >_\beta t_2$ and $P_1 >_\beta P_2$ for the transitive closure of the β -reduction on first-order terms and formulas).

We adopt the usual conventions about balancing of parentheses: arrows are right associative (it means that we write $A \multimap B \multimap C$ instead of $A \multimap (B \multimap C)$) and application is left associative (meaning we write $t_1 t_2 t_3$ instead of $(t_1 t_2) t_3$). By abuse of notation, we allow ourselves not to write the type of first and second order \forall when we can guess them from the context. We also write $!^k P$ instead of $! \dots ! P$ with k exclamation marks.

Example 2. Here are some examples of formulas of interest :

1. Leibniz's equality between two terms t_1 and t_2 of type τ

$$\forall X : [\tau], X t_1 \multimap X t_2$$

which we write it $t_1 =_\tau t_2$ in the remaining of this document.

2. The induction principle for a natural number x

$$\forall X : [\text{nat}], !(\forall y, X y \multimap X (s y)) \multimap !(X 0 \multimap X x)$$

which we write Nx where nat will be the type $\forall \alpha, (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$ of natural numbers in system \mathcal{F} and where s and 0 are first-order variables.

3. The tensor between two formulas P and Q , $\forall X, (P \multimap Q \multimap X) \multimap X$ written $P \otimes Q$.
4. And the extensionality principle

$$\forall \alpha \beta, \forall f g : \alpha \rightarrow \beta, (\forall x : \alpha, f x =_\beta g x) \multimap f =_{\alpha \rightarrow \beta} g$$

Definition 3. A *context* is an ordered list of elements of the form:

$$\alpha : \text{Type} \quad \text{or} \quad x : \tau \quad \text{or} \quad X : [\tau_1, \dots, \tau_n].$$

In the following, the beginning of the lowercase Latin alphabet a, b, \dots will designate variables of any sort and the beginning of uppercase Latin alphabet A, B, C, \dots designate *Type*, *Prop*, any type τ or something of the form $[\tau_1, \dots, \tau_n]$. We write $a \in \Gamma$, if there is an element of the form $a : _$ in Γ . A context Γ is said to be *well-formed* if “ Γ is well-formed” can be derived in the type system. A formula F (resp. a term t , resp. a type τ) is said to be well-formed in a context Γ if the sequent $\Gamma \vdash_{\text{ok}} F : \text{Prop}$ (resp. $\Gamma \vdash_{\text{ok}} t : \tau$ for some τ , resp. $\Gamma \vdash_{\text{ok}} \tau : \text{Type}$) is derivable in the type system.

$\frac{}{\emptyset \text{ is well-formed}}$	$\frac{\Gamma \text{ is well-formed}}{\Gamma, \alpha : Type \text{ is well-formed}} \alpha \notin \Gamma$
$x \notin \Gamma \frac{\Gamma \vdash \tau : Type}{\Gamma, x : \tau \text{ is well-formed}}$	$\frac{\Gamma \vdash \tau_1 : Type \dots \Gamma \vdash \tau_n : Type}{\Gamma, X : [\tau_1, \dots, \tau_n] \text{ is well-formed}} X \notin \Gamma$
$\frac{\Gamma \text{ is well-formed}}{\Gamma, a : A \vdash_{\text{ok}} a : A}$	$\frac{\Gamma \vdash_{\text{ok}} b : B}{\Gamma, a : A \vdash_{\text{ok}} b : B} a \neq b \quad \frac{\Gamma \vdash_{\text{ok}} P : Prop}{\Gamma \vdash_{\text{ok}} !P : Prop}$
$\frac{\Gamma \vdash_{\text{ok}} \tau : Type \quad \Gamma \vdash_{\text{ok}} \sigma : Type}{\Gamma \vdash_{\text{ok}} \tau \rightarrow \sigma : Type}$	$\frac{\Gamma, \alpha : Type \vdash_{\text{ok}} \tau : Type}{\Gamma \vdash_{\text{ok}} (\forall \alpha, \tau) : Type} \quad \frac{\Gamma, x : \tau \vdash_{\text{ok}} t : \sigma}{\Gamma \vdash_{\text{ok}} (\lambda x : \tau. t) : \tau \rightarrow \sigma}$
$\frac{\Gamma, \alpha : Type \vdash_{\text{ok}} t : \tau}{\Gamma \vdash_{\text{ok}} (\Lambda \alpha. t) : \forall \alpha, \tau}$	$\frac{\Gamma \vdash_{\text{ok}} f : \tau \rightarrow \sigma \quad \Gamma \vdash_{\text{ok}} a : \tau}{\Gamma \vdash_{\text{ok}} (fa) : \sigma} \quad \frac{\Gamma \vdash_{\text{ok}} f : \Lambda \alpha. \sigma \quad \Gamma \vdash_{\text{ok}} \tau : Type}{\Gamma \vdash_{\text{ok}} (f \tau) : \sigma[\tau/\alpha]}$
$\frac{\Gamma, X : [\tau_1, \dots, \tau_n] \vdash_{\text{ok}} Q : Prop}{\Gamma \vdash_{\text{ok}} (\forall X : [\tau_1, \dots, \tau_n], Q) : Prop}$	$\frac{\Gamma, x : \tau \vdash_{\text{ok}} Q : Prop}{\Gamma \vdash_{\text{ok}} (\forall x : \tau, Q) : Prop} \quad \frac{\Gamma, \alpha : Type \vdash_{\text{ok}} Q : Prop}{\Gamma \vdash_{\text{ok}} (\forall \alpha, Q) : Prop}$
$\frac{\Gamma \vdash_{\text{ok}} P : Prop \quad \Gamma \vdash_{\text{ok}} Q : Prop}{\Gamma \vdash_{\text{ok}} (P \multimap Q) : Prop}$	$\frac{\Gamma \vdash_{\text{ok}} t_1 : \tau_1 \quad \dots \quad \Gamma \vdash_{\text{ok}} t_n : \tau_n \quad \Gamma \vdash_{\text{ok}} X : [\tau_1, \dots, \tau_n]}{\Gamma \vdash_{\text{ok}} X t_1 \dots t_n : Prop}$

Type system for checking well-formedness

Example 4. These formulas are well-typed :

1. $\Gamma, x : \tau, y : \tau \vdash_{\text{ok}} x =_{\tau} y : Prop,$
2. $\Gamma, s : \text{nat} \multimap \text{nat}, 0 : \text{nat}, x : \text{nat} \vdash_{\text{ok}} Nx : Prop,$
3. $\Gamma, X : Prop, Y : Prop \vdash_{\text{ok}} X \otimes Y : Prop,$
4. $\vdash_{\text{ok}} \forall \alpha \beta, \forall f g : \alpha \rightarrow \beta, (\forall x : \alpha, f x =_{\beta} g x) \multimap f =_{\alpha \rightarrow \beta} g : Prop.$

We have shown in [5] that this simple system have numerous good properties of pure type systems (like subject reduction).

Interpretations in standard models

In this section, we build a small realizability model for our proof system which we will use later to prove the correctness with respect to the specification of the extracted proof. One of our goal is to make the model satisfy the extensionality principle, because we will need to be able to replace in our proofs higher-order terms by other extensionally equal terms.

We define the set \mathcal{P} of programs to be the set of pure λ -terms modulo β -reduction. In the following, we interpret terms in \mathcal{P} , types by partial equivalence relations (PER) on \mathcal{P} and second-order variables by sets of element in \mathcal{P} stable by extensionality (you are not allowed to consider sets which are able to distinguish terms that compute the same things). Finally, formulas are interpreted as classical formulas: all informations about linearity and exponentials are forgotten. Indeed, we forget all complexity

informations because the only purpose of model theory here is to have result about the compliance with respect to the specifications.

Definition 5. Let Γ be a well-formed context. A Γ -model consists of three partial functions recursively define below. The first one is map from type variables to PERs, the second is a map from first-order variables to \mathcal{P} and the last one is a map from second-order variables to sets of tuples of programs.

- If Γ is empty, then the only Γ -model is three empty maps.
- If Γ has the form $\Delta, x : \tau$ and if $\mathcal{M} = (\mathcal{M}_0, \mathcal{M}_1, \mathcal{M}_2)$ is a Δ -model, then for any $t \in \llbracket \tau \rrbracket_{\mathcal{M}}$, $(\mathcal{M}_0, \mathcal{M}_1[x \mapsto t], \mathcal{M}_2)$ is a Γ -model (in the following, we simply write it $\mathcal{M}[x \mapsto t]$).
- If Γ has the form $\Delta, \alpha : \text{Type}$ and if $\mathcal{M} = (\mathcal{M}_0, \mathcal{M}_1, \mathcal{M}_2)$ is a Δ -model, then for any PER R , $(\mathcal{M}_0[\alpha \mapsto R], \mathcal{M}_1, \mathcal{M}_2)$ is a Γ -model (we write it $\mathcal{M}[\alpha \mapsto R]$).
- If Γ has the form $\Delta, X : [\tau_1, \dots, \tau_n]$ and if $\mathcal{M} = (\mathcal{M}_0, \mathcal{M}_1, \mathcal{M}_2)$ is a Δ -model, then for any $E \subseteq \llbracket \tau_1 \rrbracket_{\mathcal{M}} \times \dots \times \llbracket \tau_n \rrbracket_{\mathcal{M}}$ such that E satisfy the *stability condition*

$$\text{If } (t_1, \dots, t_n) \in E \wedge t_1 \sim_{\tau_1}^{\mathcal{M}} t'_1 \wedge \dots \wedge t_n \sim_{\tau_n}^{\mathcal{M}} t'_n, \text{ then } (t'_1, \dots, t'_n) \in E$$

$(\mathcal{M}_0, \mathcal{M}_1, \mathcal{M}_2[X \mapsto E])$ is a Γ -model (we write it $\mathcal{M}[X \mapsto E]$).

Where $\sim_{\tau}^{\mathcal{M}}$ is a partial equivalence relation whose domain is written $\llbracket \tau \rrbracket_{\mathcal{M}}$ defined recursively on the structure of τ ,

- $\sim_{\alpha}^{\mathcal{M}}$ is equal to $\mathcal{M}_0(\alpha)$,
- $\sim_{\sigma \rightarrow \tau}^{\mathcal{M}}$ is defined by $t_1 \sim_{\sigma \rightarrow \tau}^{\mathcal{M}} t_2 \Leftrightarrow \forall s_1 s_2, s_1 \sim_{\sigma}^{\mathcal{M}} s_2 \Rightarrow (t_1 s_1) \sim_{\tau}^{\mathcal{M}} (t_2 s_2)$,
- $\sim_{\forall \alpha, \tau}^{\mathcal{M}} = \bigcap_{R \text{ is PER}} \sim_{\tau}^{\mathcal{M}[\alpha \mapsto R]}$.

Intuitively $t_1 \sim_{\tau}^{\mathcal{M}} t_2$ means the pure λ -terms t_1 and t_2 are of type τ and they are extensionally equivalent.

Now, we can define the interpretation $\llbracket t \rrbracket_{\mathcal{M}}$ of a first-order term t such that $\Gamma \vdash_{\text{ok}} t : \tau$ in a Γ -model \mathcal{M} to be the pure λ -term obtained by replacing all occurrences of free variables by their interpretation in \mathcal{M} and by erasing type information. And we can prove substitution lemmas.

Lemma 6. For any Γ -models \mathcal{M} ,

1. If $\Gamma, \alpha : \text{Type} \vdash_{\text{ok}} \tau : \text{Type}$ and $\Gamma \vdash_{\text{ok}} \sigma : \text{Type}$, then $\llbracket \tau[\sigma/\alpha] \rrbracket_{\mathcal{M}} = \llbracket \tau \rrbracket_{\mathcal{M}[\alpha \mapsto \sim_{\sigma}^{\mathcal{M}}]}$,
2. If $\Gamma, \alpha : \text{Type} \vdash_{\text{ok}} t : \sigma$ and $\Gamma \vdash_{\text{ok}} \tau : \tau$, then $\llbracket t[\tau/\alpha] \rrbracket_{\mathcal{M}} = \llbracket t \rrbracket_{\mathcal{M}[\alpha \mapsto \sim_{\tau}^{\mathcal{M}}]}$
3. If $\Gamma, x : \sigma \vdash_{\text{ok}} t : \tau$ and $\Gamma \vdash_{\text{ok}} s : \sigma$, then $\llbracket t[s/x] \rrbracket_{\mathcal{M}} = \llbracket t \rrbracket_{\mathcal{M}[x \mapsto \llbracket s \rrbracket_{\mathcal{M}}]}$
4. If $\Gamma \vdash_{\text{ok}} t : \tau$, $t \equiv_{\beta} t'$ and $\Gamma \vdash_{\text{ok}} t' : \tau$, then $\llbracket t \rrbracket_{\mathcal{M}} = \llbracket t' \rrbracket_{\mathcal{M}}$.

And then we can deduce an adequacy lemma about well-typed terms.

Lemma 7. If we have $\Gamma \vdash_{\text{ok}} t : \tau$ and \mathcal{M} a Γ -model, then $\llbracket t \rrbracket_{\mathcal{M}} \in \llbracket \tau \rrbracket_{\mathcal{M}}$.

Now we can define the notion of satisfiability in a model recursively on formulas' structure.

Definition 8. Let P be a formula such that $\Gamma \vdash_{\text{ok}} P : \text{Prop}$ and \mathcal{M} be a Γ -model.

- $\mathcal{M} \models X t_1 \dots t_n$ iff $(\llbracket t_1 \rrbracket_{\mathcal{M}}, \dots, \llbracket t_n \rrbracket_{\mathcal{M}}) \in \mathcal{M}(X)$,
- $\mathcal{M} \models P \multimap Q$ iff $\mathcal{M} \models P$ implies $\mathcal{M} \models Q$,
- $\mathcal{M} \models \forall X : [\tau_1, \dots, \tau_n], P$ iff for all $E \subseteq \llbracket \tau_1 \rrbracket_{\mathcal{M}} \times \dots \times \llbracket \tau_n \rrbracket_{\mathcal{M}}$ satisfying the stability condition, $\mathcal{M}[X \mapsto E] \models P$,

- $\mathcal{M} \models \forall x : \tau, P$ iff for all $t \in \llbracket \tau \rrbracket_{\mathcal{M}}$, $\mathcal{M}[x \mapsto t] \models P$,
- $\mathcal{M} \models \forall \alpha, P$ iff for all PER R on \mathcal{P} , $\mathcal{M}[\alpha \mapsto R] \models P$,
- $\mathcal{M} \models !P$ iff $\mathcal{M} \models P$.

If E is a set of formulas well-formed in Γ , for all Γ -model \mathcal{M} , we write $\mathcal{M} \models E$ for meaning that $\mathcal{M} \models Q$ for all $Q \in E$. And if T is another set of formulas well-formed in Γ , we write $T \models_{\Gamma} E$ if for all Γ -model \mathcal{M} , $\mathcal{M} \models T$ implies $\mathcal{M} \models E$ (and we write $T \models_{\Gamma} P$ in place of $T \models_{\Gamma} \{P\}$).

Lemma 9. *The formulas are unable to distinguish extensionally equivalent programs: for any formula P such that $\Gamma, x_1 : \tau_1, \dots, x_n : \tau_n \vdash_{ok} P : Prop$ and any Γ -model \mathcal{M} the set*

$$\{(t_1, \dots, t_n) \in \llbracket \tau_1 \rrbracket \times \dots \times \llbracket \tau_n \rrbracket \mid \mathcal{M}[x_1 \mapsto t_1, \dots, x_n \mapsto t_n] \models P\}$$

satisfies the stability condition.

Lemma 10. *For any Γ -model \mathcal{M} ,*

1. *If $\Gamma, \alpha : Type \vdash_{ok} P : Prop$ and $\Gamma \vdash_{ok} \tau : Type$, then $\mathcal{M} \models P[\tau/\alpha] \Leftrightarrow \mathcal{M}[\alpha \mapsto \sim_{\tau}^{\mathcal{M}}] \models P$,*
2. *If $\Gamma, x : \tau \vdash_{ok} P : Prop$ and $\Gamma \vdash_{ok} t : \tau$, then $\mathcal{M} \models P[t/x] \Leftrightarrow \mathcal{M}[x \mapsto \llbracket t \rrbracket_{\mathcal{M}}] \models P$,*
3. *If $\Gamma, X : [\tau_1, \dots, \tau_n] \vdash_{ok} P : Prop$ and $\Gamma, x_1 : \tau_1, \dots, x_n : \tau_n \vdash_{ok} Q : Prop$, then*

$$\mathcal{M} \models P[Q/X x_1 \dots, x_n] \Leftrightarrow \mathcal{M}[X \mapsto E] \models P$$

where

$$E = \{(t_1, \dots, t_n) \in \llbracket \tau_1 \rrbracket \times \dots \times \llbracket \tau_n \rrbracket \mid \mathcal{M}[x_1 \mapsto t_1, \dots, x_n \mapsto t_n] \models Q\},$$

4. *If $\Gamma \vdash_{ok} P : Prop$, $P \equiv_{\beta} P'$ and $\Gamma \vdash_{ok} P' : Prop$, then $\mathcal{M} \models P \Leftrightarrow \mathcal{M} \models P'$.*

Lemma 11. *If $\Gamma \vdash_{ok} t_1 : \tau$, $\Gamma \vdash_{ok} t_2 : \tau$ and \mathcal{M} is a Γ -model, then $\mathcal{M} \models t_1 =_{\tau} t_2 \Leftrightarrow \llbracket t_1 \rrbracket_{\mathcal{M}} \sim_{\tau}^{\mathcal{M}} \llbracket t_2 \rrbracket_{\mathcal{M}}$.*

Proof.

- $\mathcal{M} \models t_1 =_{\tau} t_2 \Rightarrow \llbracket t_1 \rrbracket_{\mathcal{M}} \sim_{\tau}^{\mathcal{M}} \llbracket t_2 \rrbracket_{\mathcal{M}}$: Let $E = \{t \in \llbracket \tau \rrbracket_{\mathcal{M}} \mid \llbracket t_1 \rrbracket_{\mathcal{M}} \sim_{\tau}^{\mathcal{M}} t\}$ be the equivalence class of $\llbracket t_1 \rrbracket_{\mathcal{M}}$ (as such E satisfy the stability condition). If $\mathcal{M} \models t_1 =_{\tau} t_2$, then $\mathcal{M}[X \mapsto E] \models X t_1 \multimap X t_2$ which means that $\llbracket t_1 \rrbracket_{\mathcal{M}} \in E$ -which is true- implies $\llbracket t_2 \rrbracket_{\mathcal{M}} \in E$ which means that $\llbracket t_1 \rrbracket_{\mathcal{M}} \sim_{\tau}^{\mathcal{M}} \llbracket t_2 \rrbracket_{\mathcal{M}}$.
- $\llbracket t_1 \rrbracket_{\mathcal{M}} \sim_{\tau}^{\mathcal{M}} \llbracket t_2 \rrbracket_{\mathcal{M}} \Rightarrow \mathcal{M} \models t_1 =_{\tau} t_2$: Suppose $\llbracket t_1 \rrbracket_{\mathcal{M}} \sim_{\tau}^{\mathcal{M}} \llbracket t_2 \rrbracket_{\mathcal{M}}$, then for all $E \subseteq \llbracket \tau \rrbracket_{\mathcal{M}}$ satisfying the stability condition, we have $\llbracket t_1 \rrbracket_{\mathcal{M}} \in E$ implies $\llbracket t_2 \rrbracket_{\mathcal{M}} \in E$ or in other words $\mathcal{M}[X \mapsto E] \models X t_1 \multimap X t_2$. And therefore, we obtain $\mathcal{M} \models t_1 =_{\tau} t_2$.

□

Definition 12. Suppose we have $\Gamma \vdash_{ok} P_1 : Prop$, $\Gamma \vdash_{ok} t_1 : \tau$ and $\Gamma \vdash_{ok} t_2 : \tau$, we say that $P_1 \xrightarrow{t_1=t_2} P_2$ if there exists a formula Q such that $\Gamma, x : \tau \vdash_{ok} Q : Prop$, $P_1 \equiv Q[t_1/x]$ and $P_2 \equiv Q[t_2/x]$.

Lemma 13.

If $\mathcal{M} \models t_1 =_{\tau} t_2$ and $P_1 \xrightarrow{t_1=t_2} P_2$ then $\mathcal{M} \models P_1 \Rightarrow \mathcal{M} \models P_2$.

Proof. Suppose $P_1 \equiv Q[t_1/x]$ and $P_2 \equiv Q[t_2/x]$. Let E be the set $\{t \in \llbracket \tau \rrbracket_{\mathcal{M}} \mid \mathcal{M}[x \mapsto t] \models Q\}$. Since $\mathcal{M} \models t_1 =_{\tau} t_2$, we have that $\mathcal{M}[X \mapsto E] \models X t_1 \multimap X t_2$ which is equivalent to $\mathcal{M}[x \mapsto \llbracket t_1 \rrbracket_{\mathcal{M}}] \models Q$ implies $\mathcal{M}[x \mapsto \llbracket t_2 \rrbracket_{\mathcal{M}}] \models Q$, or $\mathcal{M} \models P_1$ implies $\mathcal{M} \models P_2$, or $\mathcal{M} \models P_1 \multimap P_2$. □

Theorem 14.

Theses models satisfy the extensionality principle :

$$\mathcal{M} \models \forall \alpha \beta, \forall f g : \alpha \rightarrow \beta, (\forall x : \alpha, f x =_{\beta} g x) \multimap f =_{\alpha \rightarrow \beta} g.$$

Proof. It is a consequence of the last two lemmas.

- The last one gives us that

$$\mathcal{M} \models \forall \alpha \beta, \forall f g : \alpha \rightarrow \beta, (\forall x : \alpha, f x =_{\beta} g x) \multimap (\forall x y : \alpha, x =_{\alpha} y \multimap f x =_{\beta} g y).$$

- Therefore we are left to prove that $\mathcal{M} \models \forall \alpha \beta, \forall f g : \alpha \rightarrow \beta, (\forall x y : \alpha, x =_{\alpha} y \multimap f x =_{\beta} g y) \multimap f =_{\alpha \rightarrow \beta} g$. Let R_{α} and R_{β} be two PER, $t_1, t_2 \in \llbracket \alpha \rightarrow \beta \rrbracket_{\mathcal{M}[\alpha \mapsto R_{\alpha}, \beta \mapsto R_{\beta}]}$. Suppose $\mathcal{M}[\alpha \mapsto R_{\alpha}, \beta \mapsto R_{\beta}, f \mapsto t_1, g \mapsto t_2] \models \forall x y : \alpha, x =_{\alpha} y \multimap f x =_{\beta} g y$, we need to prove that, $\mathcal{M}[\alpha \mapsto R_{\alpha}, \beta \mapsto R_{\beta}, f \mapsto t_1, g \mapsto t_2] \models f =_{\alpha \rightarrow \beta} g$ or equivalently that $t_1 \sim_{\alpha \rightarrow \beta}^{\mathcal{M}[\alpha \mapsto R_{\alpha}, \beta \mapsto R_{\beta}]} t_2$, which is also equivalent to the fact that for all $(a_1, a_2) \in R_{\alpha}$, $((t_1 a_1), (t_2 a_2)) \in R_{\beta}$ which is exactly $\mathcal{M}[\alpha \mapsto R_{\alpha}, \beta \mapsto R_{\beta}, f \mapsto t_1, g \mapsto t_2] \models \forall x y : \alpha, x =_{\alpha} y \multimap f x =_{\beta} g y$.

□

Projecting formulas toward types

In order to write the rules of our proof system in the next section, we are going need to have way to project second-order formulas toward types.

Definition 15. Given a formula F , we define the type F^{-} recursively built from F in the following way.

$$(X t_1 \dots t_n)^{-} \equiv \alpha_X \quad (A \multimap B)^{-} \equiv A^{-} \rightarrow B^{-} \quad (\forall \alpha, F)^{-} \equiv F^{-} \quad (\forall x : \alpha, F)^{-} \equiv F^{-} \quad (!F)^{-} \equiv !F^{-} \\ (\forall X : [\tau_1, \dots, \tau_n], F)^{-} \equiv \forall \alpha_X, F^{-}.$$

Lemma 16. *If $\Gamma \vdash_{ok} A : Prop$, $\Gamma^* \vdash_{ok} A^{-} : Type$ where Γ^* is obtained from Γ by replacing occurrences of “ $X : [\tau_1, \dots, \tau_n]$ ” by “ $\alpha_X : Type$ ” and letting others unchanged.*

Example 17.

- $(t_1 =_{\tau} t_2)^{-} \equiv \forall \alpha, \alpha \multimap \alpha \equiv \text{unit}.$
- $(Nx)^{-} \equiv (\forall X : [\text{nat}], !(\forall y, X y \multimap X (s y)) \multimap !(X 0 \multimap X x))^{-} \equiv \forall \alpha, (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha \equiv \text{nat},$

2 The proof system

Sequents are of the form $\Gamma; \Delta \vdash t : P$ where Γ is a context (see definition 3), Δ is an unordered set of assignments of the form $x : Q$ where t is a first-order term, x a first-order variable and P and Q are formulas. Our proof system has two parameters:

- A well-formed typing context Σ of types of functions we want to implement. In this paper, we use the set

$$\Sigma = \{ \quad 0 : \text{nat}, s : \text{nat} \rightarrow \text{nat}, \text{pred} : \text{nat} \rightarrow \text{nat}, \text{mult} : \text{nat} \rightarrow \text{nat} \rightarrow \text{nat}, \\ \text{minus} : \text{nat} \rightarrow \text{nat} \rightarrow \text{nat}, \text{plus} : \text{nat} \rightarrow \text{nat} \rightarrow \text{nat}, \\ \text{sum} : (\text{nat} \rightarrow \text{nat}) \rightarrow \text{nat} \rightarrow \text{nat}, \text{prod} : (\text{nat} \rightarrow \text{nat}) \rightarrow \text{nat} \rightarrow \text{nat} \quad \}.$$

- A set \mathcal{H} of equational formulas of the form $\forall x_1 : \tau_1, \dots, \forall x_n : \tau_n, t_1 =_\tau t_2$ well-typed in Σ . In this paper, we take \mathcal{H} to be the intersection of all sets T of formulas of this form such that $\mathcal{H}_0 \models_\Sigma T$ where \mathcal{H}_0 is the set below.

$$\mathcal{H}_0 \models \{$$

$\forall n : \text{nat},$	0	$=_{\text{nat}}$	$\Lambda \alpha. \lambda f : \alpha \rightarrow \alpha. x : \alpha. x$,
$\forall xy : \text{nat},$	sn	$=_{\text{nat}}$	$\Lambda \alpha. \lambda f : \alpha \rightarrow \alpha. x : \alpha. n \alpha f(fx)$,
$\forall x : \text{nat},$	$plus\ x\ (s\ y)$	$=_{\text{nat}}$	$s\ (plus\ x\ y)$,
$\forall xy : \text{nat},$	$plus\ x\ 0$	$=_{\text{nat}}$	x	,
$\forall xy : \text{nat},$	$mult\ x\ (s\ y)$	$=_{\text{nat}}$	$plus\ x\ (mult\ x\ y)$,
$\forall x : \text{nat},$	$mult\ x\ 0$	$=_{\text{nat}}$	0	,
$\forall x : \text{nat},$	$pred\ (s\ x)$	$=_{\text{nat}}$	x	,
$\forall x : \text{nat},$	$pred\ 0$	$=_{\text{nat}}$	0	,
$\forall xy : \text{nat},$	$minus\ x\ (s\ y)$	$=_{\text{nat}}$	$pred\ (minus\ x\ y)$,
$\forall x : \text{nat},$	$minus\ x\ 0$	$=_{\text{nat}}$	x	,
$\forall x : \text{nat}, \forall f : \text{nat} \rightarrow \text{nat},$	$sum\ f\ (s\ x)$	$=_{\text{nat}}$	$plus\ (sum\ f\ x)\ (f\ x)$,
$\forall f : \text{nat} \rightarrow \text{nat},$	$sum\ f\ 0$	$=_{\text{nat}}$	0	,
$\forall x : \text{nat}, \forall f : \text{nat} \rightarrow \text{nat},$	$prod\ f\ (s\ x)$	$=_{\text{nat}}$	$mult\ (prod\ f\ x)\ (f\ x)$,
$\forall f : \text{nat} \rightarrow \text{nat},$	$prod\ f\ 0$	$=_{\text{nat}}$	$s\ 0$	}

$\text{AXIOM} \frac{\Sigma, \Gamma \vdash_{\text{ok}} P : \text{Prop}}{\Gamma; x : P \vdash x : P}$	$x \notin \Delta$	$\text{WEAKENING} \frac{\Gamma; \Delta \vdash t : Q \quad \Sigma, \Gamma \vdash_{\text{ok}} P : \text{Prop}}{\Gamma; \Delta, x : P \vdash t : Q}$
$\text{APPLICATION} \frac{\Gamma; \Delta_1 \vdash t_1 : P \multimap Q \quad \Gamma; \Delta_2 \vdash t_2 : P}{\Gamma; \Delta_1, \Delta_2 \vdash (t_1 t_2) : Q}$	$\text{ABSTRACTION} \frac{\Gamma; \Delta, x : P \vdash t : Q}{\Gamma; \Delta \vdash \lambda x : P^-. t : P \multimap Q}$	
$\text{PROMOTION} \frac{\Gamma; \Delta_1 \vdash t_1 : !P_1 \quad \dots \quad \Gamma; \Delta_n \vdash t_n : !P_n \quad \Gamma; x_1 : P_1, \dots, x_n : P_n \vdash t : P}{\Gamma; \Delta_1, \dots, \Delta_n \vdash t[t_1/x_1, \dots, t_n/x_n] : !P}$		
$\text{CONTRACTION} \frac{\Gamma; \Delta, x : !P, x : !P \vdash t : Q}{\Gamma; \Delta, x : !P \vdash t : Q}$	$\forall_\alpha\text{-INTRO} \frac{\Gamma, \alpha : \text{Type}; \Delta \vdash t : P}{\Gamma; \Delta \vdash t : \forall \alpha, P}$	
$\forall_1\text{-INTRO} \frac{\Gamma, x : \tau; \Delta \vdash t : P}{\Gamma; \Delta \vdash t : \forall x : \tau, P}$	$\forall_2\text{-INTRO} \frac{\Gamma, X : [\tau_1, \dots, \tau_n]; \Delta \vdash t : P}{\Gamma; \Delta \vdash (\Lambda \alpha_X. t) : \forall X : [\tau_1, \dots, \tau_n], P}$	
$\forall_\alpha\text{-ELIM} \frac{\Gamma; \Delta \vdash t : \forall \alpha, P \quad \Sigma, \Gamma \vdash_{\text{ok}} \tau : \text{Type}}{\Gamma; \Delta \vdash t : P[\tau/\alpha]}$	$\forall_1\text{-ELIM} \frac{\Gamma; \Delta \vdash t : \forall x : \tau, P \quad \Sigma, \Gamma \vdash_{\text{ok}} a : \tau}{\Gamma; \Delta \vdash t : P[a/x]}$	
$\forall_2\text{-ELIM} \frac{\Gamma; \Delta \vdash t : \forall X : [\tau_1, \dots, \tau_n], P \quad \Sigma, \Gamma, x_1 : \tau_1, \dots, x_n : \tau_n \vdash_{\text{ok}} Q : \text{Prop}}{\Gamma; \Delta \vdash (t Q^-) : P[Q/X x_1 \dots x_n]}$		
$\text{EQUALITY} \frac{\mathcal{H} \models_{\Sigma, \Gamma} t_1 =_\tau t_2 \text{ and } P_1 \xrightarrow{t_1 =_\tau t_2} P_2 \quad \Gamma; \Delta \vdash t : P_1}{\Gamma; \Delta \vdash t : P_2}$		

The proof system parametrized by Σ and \mathcal{H}

The following lemma gives us the type of proof-terms.

Lemma 18. *If $\Gamma; x_1 : P_1, \dots, x_n : P_n \vdash t : P$, then $\Gamma^*, x_1 : P_1^-, \dots, x_n : P_n^- \vdash_{\text{ok}} t : P^-$.*

And this one tells us that our proof system is well-behaved with respect to our notion of model.

Lemma 19. (*Adequacy lemma*)

If $\Gamma; x_1 : P_1, \dots, x_n : P_n \vdash t : P$, then $\mathcal{H} \cup \{P_1, \dots, P_n\} \models_{\Gamma} P$.

Proof. The proof consists of an induction on the structure of the proof $\Gamma; x_1 : P_1, \dots, x_n : P_n \vdash t : P$ and an intensive use of substitution lemmas. \square

A simple realizability theory

Definition 20. Given a formula F and a term t , we can recursively define the formula written $t \Vdash F$ upon the structure of F in the following way.

- $t \Vdash X t_1 \dots t_n \equiv X t_1 \dots t_n t$,
- $t \Vdash P \multimap Q \equiv \forall x : P^-, x \Vdash P \multimap (t x) \Vdash Q$,
- $t \Vdash \forall X : [\tau_1, \dots, \tau_n], P \equiv \forall \alpha_X, \forall X : [\tau_1, \dots, \tau_n, \alpha_X], t \alpha_X \Vdash P$,
- $t \Vdash \forall x : \tau, P \equiv \forall x : \tau, t \Vdash P$,
- $t \Vdash \forall \alpha, P \equiv \forall \alpha, t \Vdash P$,
- $t \Vdash !P \equiv !(t \Vdash P)$.

Lemma 21. For any formula P and any context Γ and any first-order term t ,

$$\left. \begin{array}{l} \Gamma \vdash_{ok} P : Prop \\ \Gamma^* \vdash_{ok} t : P^- \end{array} \right\} \Rightarrow \Gamma^- \vdash_{ok} (t \Vdash P) : Prop$$

where Γ^- is obtained from Γ by replacing each occurrence of “ $X : [\tau_1, \dots, \tau_n]$ ” by “ $\alpha_X : Type, X : [\tau_1, \dots, \tau_n, \alpha_X]$ ” (and $\Gamma^* \subseteq \Gamma^-$ as in lemma 16).

Lemma 22. (*Adequacy lemma for realizers*)

If $\Gamma; x_1 : P_1, \dots, x_n : P_n \vdash t : P$, then

$$\Gamma, x_1 : P_1^-, \dots, x_n : P_n^-; x_1 : (x_1 \Vdash P_1), \dots, x_n : (x_n \Vdash P_n) \vdash t : (t \Vdash P).$$

Proof. It is a consequence of the good “applicative behavior” of realizability. The result comes easily with an induction on the structure of proof of $\Gamma; x_1 : P_1, \dots, x_n : P_n \vdash t : P$. \square

Programming with proofs

Definition 23. Let D be a formula such that $\Gamma, x : \tau \vdash_{ok} Dx : Prop$ for some τ . We say that D is *data type* of parameter x of type D^- relatively to a Γ -model \mathcal{M} if we have :

1. $\mathcal{M} \models \forall r x : D^-, (r \Vdash D) \multimap r =_{\tau} x$,
2. $\mathcal{M} \models \forall x : D^-, x \Vdash D$ (or equivalently the converse $\forall r x : D^-, r =_{\tau} x \multimap (r \Vdash D)$ of 1.)

We simply say that Dy is a data type in \mathcal{M} , if D is a data type of parameter y relatively to \mathcal{M} and for any term t such that $\Gamma \vdash_{ok} t : D^-$, we write Dt instead of $D[t/y]$.

Lemma 24. Nx is a data type in all Σ -models.

Proof. The proof is similar that the one for FA_2 in [4]. \square

Lemma 25. *If Ax and Bx are two data types in a Γ -model \mathcal{M} , so is $Ff \equiv \forall x : A^-, Ax \multimap B(fx)$.*

Proof. We have to verify the two conditions of the definition.

1. If \mathcal{M}' is a $\Gamma, r : A^- \rightarrow B^-, f : A^- \rightarrow B^-$ -model such that $\mathcal{M}' \models r \Vdash Ff$. Since $r \Vdash Ff \equiv \forall s x, s \Vdash Ax \multimap (rs) \Vdash B(fx)$ and by invoking the second condition for A and the first for B we have $\mathcal{M} \models \forall s x, s =_{A^-} x \multimap (rs) =_{B^-} (fx)$ which is equivalent by extensionality to $\mathcal{M} \models r =_{A^- \rightarrow B^-} f$.
2. Let \mathcal{M}' be a $\Gamma, f : A^- \rightarrow B^-$ -model, we have to prove that $\mathcal{M}' \models f \Vdash Ff$ or equivalently that $\mathcal{M} \models \forall r x : A^- x, r \Vdash Ax \multimap (fr) \Vdash B(fx)$. But according to the first condition for A it is stronger that $\mathcal{M} \models \forall r x : A^- x, r =_{A^-} x \multimap (fr) \Vdash B(fx)$ which is implied the second condition for B .

□

The following theorem state that if we can find a model \mathcal{M} satisfying \mathcal{H} (informally it means that we know our specifications to be implementable), then the program t extracted from the proof of a formula stating that a function f is provably total implements this function.

Theorem 26. *Let $D_1 x_1, \dots, D_n x_n$, and D be $n+1$ data types. If $\Gamma \vdash_{ok} f : D_1^- \rightarrow \dots \rightarrow D_n^- \rightarrow D^-$ If*

$$\Gamma; \vdash t : \forall x_1 : D_1^-, \dots, x_n : D_n^-, D_1 x_1 \multimap \dots \multimap D_n x_n \multimap D(fx_1 \dots x_n),$$

then for all $\Sigma, \Gamma, f : D_1^- \rightarrow \dots \rightarrow D_n^- \rightarrow D^-$ -model \mathcal{M} such $\mathcal{M} \models \mathcal{H}$,

$$\mathcal{M} \models t =_{D_1^- \rightarrow \dots \rightarrow D_n^- \rightarrow D^-} f.$$

Proof. By lemma 22 we have $\Gamma; \vdash t \Vdash D_1 x_1 \multimap \dots \multimap D_n x_n \multimap D(fx_1 \dots x_n)$ which is equivalent to

$$\Gamma; \vdash \forall r_1 x_1 : D_1^-, \dots, \forall r_n x_n : D_n^-, r_1 \Vdash D_1 x_1 \multimap \dots \multimap r_n \Vdash D_n x_n \multimap (t r_1 \dots r_n) \Vdash D(fx_1 \dots x_n)$$

by lemma 19 we have

$$\mathcal{M} \models \forall r_1 x_1 : D_1^-, \dots, \forall r_n x_n : D_n^-, r_1 \Vdash D_1 x_1 \multimap \dots \multimap r_n \Vdash D_n x_n \multimap (t r_1 \dots r_n) \Vdash D(fx_1 \dots x_n)$$

but since every one is a data type we obtain

$$\mathcal{M} \models \forall r_1 x_1 : D_1^-, \dots, \forall r_n x_n : D_n^-, r_1 =_{D_1^-} x_1 \multimap \dots \multimap r_n =_{D_n^-} x_n \multimap (t r_1 \dots r_n) =_{D^-} (fx_1 \dots x_n)$$

which is equivalent to $\mathcal{M} \models t =_{D_1^- \rightarrow \dots \rightarrow D_n^- \rightarrow D^-} f$.

□

3 Elementary Time Characterisation

Correctness

We describe here how we can bring our system back toward Elementary Affine Logic in order to prove that extracted programs are elementary bounded. In this section, we will consider the grammar of second-order elementary logic which is basically a linear version of system \mathcal{F} types.

$$\tau, \sigma, \dots := \alpha \mid \forall \alpha, \tau \mid \sigma \multimap \tau \mid !\tau$$

Definition 27. Given a formula F , we define the type F° recursively built from F in the following way.

$$(X t_1 \dots t_n)^\circ = \alpha_X \quad (A \multimap B)^\circ = A^\circ \multimap B^\circ \quad (\forall \alpha, F)^\circ = F^\circ \quad (\forall x : \alpha, F)^\circ = F^\circ \quad (!F)^\circ = !F^\circ$$

$$(\forall X : [\tau_1, \dots, \tau_n], F)^\circ = \forall \alpha_X, F^\circ.$$

We map the rules of our system by removing first-order with our map $\cdot \mapsto \cdot^\circ$, the rules of equality, introduction and elimination for first-order \forall and type \forall then become trivial. We also erase some type information on typed terms in order to obtain the following *à la church* type system which is known as *elementary affine logic*.

$\frac{}{x : \tau \vdash_{\text{eal}} x : \tau} \text{AXIOM}$	$\frac{\Delta \vdash_{\text{eal}} t : \sigma}{\Delta, x : \tau \vdash_{\text{eal}} t : \sigma} \text{WEAKENING}$	$\frac{\Delta, x : !\sigma, x : !\sigma \vdash_{\text{eal}} t : \tau}{\Delta, x : !\sigma \vdash_{\text{eal}} t : \tau} \text{CONTRACTION}$
$\frac{\Delta_1 \vdash t_1 : !\tau_1 \quad \dots \quad \Delta_n \vdash t_n : !\tau_n \quad x_1 : \tau_1, \dots, x_n : \tau_n \vdash_{\text{eal}} t : \tau}{\Delta_1, \dots, \Delta_n \vdash_{\text{eal}} t[t_1/x_1, \dots, t_n/x_n] : !\tau} \text{PROMOTION}$		
$\frac{\Delta_1 \vdash_{\text{eal}} s : \tau \multimap \sigma \quad \Delta_2 \vdash_{\text{eal}} t : \tau}{\Delta_1, \Delta_2 \vdash_{\text{eal}} (s t) : \sigma} \text{APPLICATION}$		$\frac{\Delta, x : \sigma \vdash_{\text{eal}} t : \tau}{\Delta \vdash_{\text{eal}} (\lambda x : \sigma. t) : \sigma \multimap \tau} \text{ABSTRACTION}$
$\alpha \notin \Delta \frac{\Delta \vdash_{\text{eal}} t : \tau}{\Delta \vdash_{\text{eal}} t : \forall \alpha, \tau} \forall\text{-INTRO} \quad \frac{\Delta \vdash_{\text{eal}} t : \forall \alpha, \tau}{\Delta \vdash_{\text{eal}} t : \tau[\sigma/\alpha]} \forall\text{-ELIM}$		
Elementary Affine Logic		

We use this translation from our type system to elementary affine logic to obtain the following lemma.

Lemma 28. *If $\Gamma; \Delta \vdash_{\text{ok}} t : P$, then $\Delta^\circ \vdash_{\text{eal}} \bar{t} : P^\circ$ where \bar{t} is the pure term obtained by removing type information from t and Δ° is obtained by sending $x : P$ to $x : P^\circ$.*

The data type Nx representing integers is sent to $(Nx)^\circ = \forall \alpha, !(\alpha \multimap \alpha) \multimap !(\alpha \multimap \alpha)$ (denoted N°).

Definition 29. We say that a program $t \in \mathcal{P}$ represent a (set-theoretical) total function f if for all integers m_1, \dots, m_n , the term $(t \lceil m_1 \rceil \dots \lceil m_n \rceil)$ may be normalized to the church numeral $\lceil f(m_1, \dots, m_n) \rceil$. We say that $t \in \mathcal{E}$ if it represents a total function f belonging to the set of elementary computable functions (where $\lceil m \rceil$ is the m -th Church integer).

The following lemma is a bit of a folklore result. The closest reference would be the appendix of [3].

Lemma 30. *If $\vdash_{\text{eal}} t : !^{k_1} N^\circ \multimap \dots \multimap !^{k_n} N^\circ \multimap !^k N^\circ$ then $t \in \mathcal{E}$.*

Proof. (very rough sketch) You can bring the normalization of $(t \lceil m_1 \rceil \dots \lceil m_n \rceil)$ back to the normalization of a proof net corresponding to the proof tree that $\vdash_{\text{eal}} (t \lceil m_1 \rceil \dots \lceil m_n \rceil) : !^k N^\circ$. Promotion rules are represented as boxes in the proof net. These boxes stratify the proof net in the sense that we can define the *depth* of a node to be the number of boxes containing this node. And the depth of the net is the maximal depth of its nodes. If N is the size of the proof net, then there is a clever strategy to eliminate all cuts at a given depth (without changing the depth) by multiplying the size of the net by at most 2^N . We therefore obtain the exponential tower by iterating this process for each depth. \square

Finally by combining the last two lemmas, we prove the desired correctness theorem.

Theorem 31. *If we have*

$$\Gamma, f : \text{nat} \rightarrow \dots \rightarrow \text{nat}; \vdash t : \forall x_1 : \text{nat} \dots \forall x_n : \text{nat}, !^{k_1} N x_1 \multimap \dots \multimap !^{k_n} N x_k \multimap !^k N(f x_1 \dots x_n)$$

then $\bar{t} \in \mathcal{E}$ where \bar{t} is the untyped λ -term obtained by erasing type information from t .

Completeness

In this section we give two proofs of the fact that all elementary recursive functions may be extracted from a proof of totality.

In order to ease the reading on paper, we omit term annotations (the “ $x :$ ” in Δ and “ $t :$ ” on the right-hand side of the symbol \vdash) since, given a proof tree, theses decorations are unique up to renaming of variables. We also allow ourselves to let the typing context Γ and proofs of the typing sequents \vdash_{ok} implicit. Theses three derivable rules will be very useful in the following.

Lemma 32. *These rules are derivable:*

$$\frac{\Delta \vdash A}{!\Delta \vdash !A} \quad \frac{\Delta, A, B \vdash C}{\Delta, A \otimes B \vdash C} \quad \frac{\Delta_1 \vdash A \quad \Delta_2 \vdash B}{\Delta_1, \Delta_2 \vdash A \otimes B}$$

First proof of completeness: using the completeness of EAL

The following theorem gives us a link between typable terms in *ELL* and provably total functions in our system. And if we admit the completeness of EAL, it gives us directly that all elementary recursive functions may be extracted from a proof of totality.

Theorem 33. *Let t such that $\vdash_{\text{eal}} t : \text{nat} \multimap \dots \multimap \text{nat} \multimap !^k \text{nat}$, then*

$$\vdash \forall x_1 \dots x_n, N x_1 \multimap \dots \multimap N x_n \multimap !^{k+1} N (t x_1 \dots x_n).$$

Proof. Let N be the formula $\forall X, !(X \multimap X) \multimap !(X \multimap X)$. We have a natural embedding of EAL in our system by translating type variables to second-order variables. Therefore, we have $\vdash t : N \multimap \dots \multimap N \multimap !^k N$ and then $\vdash (t \Vdash N \multimap \dots \multimap N \multimap !^k N) (*)$. We are going to need the two simple lemmas below:

1. We have $\vdash \forall r, (r \Vdash N) \multimap N(r \text{nat } s 0)$.

The idea of the proof is that $(r \Vdash N)$ is equal to

$$\forall \alpha, \forall X : [\alpha], \forall f : \alpha, !(\forall y, X y \multimap X (f y)) \multimap !(\forall z, X z \multimap X (r \alpha f z))$$

and by taking $\alpha = \text{nat}$, $y = s$ and $z = 0$, we obtain $N(r \text{nat } s 0)$.

2. And we have $\vdash \forall r, N r \multimap !(r \Vdash N)$.

Let H be $!(\forall y, y \Vdash N \multimap (s y) \Vdash N) \multimap !(0 \Vdash N \multimap r \Vdash N)$.

$$\frac{\frac{\frac{0 \Vdash N \multimap r \Vdash N \vdash (0 \Vdash N) \multimap (r \Vdash N)}{0 \Vdash N \multimap r \Vdash N \vdash r \Vdash N}}{!(0 \Vdash N \multimap r \Vdash N) \vdash !(r \Vdash N)}}{\vdash !(0 \Vdash N \multimap r \Vdash N) \multimap !(r \Vdash N)} \quad \frac{\frac{\frac{N r \vdash N r}{N r \vdash H}}{\vdash !(\forall y, y \Vdash N \multimap (s y) \Vdash N)}}{\vdash !(\forall y, y \Vdash N \multimap (s y) \Vdash N)}}{\frac{N r \vdash !(r \Vdash N)}{\vdash \forall r, N r \multimap !(r \Vdash N)}} \quad \frac{\frac{\frac{\vdots}{\pi_1}}{\vdash 0 \Vdash N}}{\vdash \forall r, N r \multimap !(r \Vdash N)} \quad \frac{\frac{\frac{\vdots}{\pi_2}}{\vdash \forall y, y \Vdash N \multimap (s y) \Vdash N}}{\vdash !(\forall y, y \Vdash N \multimap (s y) \Vdash N)}}{\vdash \forall r, N r \multimap !(r \Vdash N)}$$

where π_1 and π_2 use the rule EQUALITY with

$$\mathcal{H} \models_{\alpha : \text{Type}, f : \alpha \rightarrow \alpha, z : \alpha} (0 \alpha f z) =_{\alpha} z \text{ and } \mathcal{H} \models_{y : \text{nat}, \alpha : \text{Type}, f : \alpha \rightarrow \alpha, z : \alpha} (s y \alpha f z) =_{\alpha} (y \alpha f (f z)).$$

Now to prove the sequent $\vdash \forall x_1 \dots x_n, N x_1 \multimap \dots \multimap N x_n \multimap !^{k+1} N(t x_1 \dots x_n)$, it is enough to find a proof of $\vdash \forall x_1 \dots x_n, !N x_1 \multimap \dots \multimap !N x_n \multimap !^k N(t x_1 \dots x_n)$ (using the PROMOTION rule). By invoking 2, we just have to prove that $\vdash \forall x_1 \dots x_n, (x_1 \Vdash N) \multimap \dots \multimap (x_n \Vdash N) \multimap !^k N(t x_1 \dots x_n)$ and then by invoking 1, we have to prove $\vdash \forall x_1 \dots x_n, (x_1 \Vdash N) \multimap \dots \multimap (x_n \Vdash N) \multimap (t x_1 \dots x_n) \Vdash !^k N$ which is equivalent to (*). \square

Second proof of completeness : encoding Kalmar's functions

The characterization due to Kalmar [7] states that elementary recursive functions is the smallest class of functions containing some base functions (constants, projections, addition, multiplication and subtraction) and stable by a composition scheme, by bounded sum and bounded product. In the remaining of the document, we will show how we can implement this functions and these schemes in our system.

- It is very easy to find a proof of $\vdash N 0$ and a proof $\vdash \forall x, N x \multimap N(s x)$. We can obtain a proof $\vdash N(s 0)$ by composing them.
- The following proof gives us the addition (in order to make it fit we cut it in two bits, and the \vdash mean the proof can be easily completed). We use “ $x + y$ ” as a notation for the term (*plus* xy).

$$\begin{array}{c}
 \vdots \\
 \hline
 \pi \quad \frac{N y, !F \vdash ! (X 0 \multimap X y)}{\quad} \quad \frac{X y \multimap X(x + y), X 0 \multimap X y \vdash X 0 \multimap X(x + y)}{\quad} \\
 \hline
 \frac{N x, N y, !F, !F \vdash ! (X 0 \multimap X(s x))}{\vdash \forall xy : \text{nat}, N x \multimap N y \multimap N(x + y)} \\
 \\
 \frac{\frac{\frac{N x \vdash N x}{N x \vdash !(\forall z, X(z + y) \multimap X((s z) + y)) \multimap ! (X(0 + y) \multimap X(x + y))}}{N x \vdash !(\forall z, X(z + y) \multimap X((s z) + y)) \multimap ! (X y \multimap X(x + y))}}{\quad} \quad \frac{\vdots}{!F \vdash !(\forall z, X(z + y) \multimap X(s(z + y)))} \\
 \hline
 \frac{N x, !F \vdash ! (X y \multimap X(x + y))}{\pi}
 \end{array}$$

Note that we have used in the left branch the EQUALITY rule with $\mathcal{H} \models \forall xy, (s x) + y = s(x + y)$ and $\mathcal{H} \models \forall y, 0 + y = y$. We extract the usual λ -term for addition $\lambda nm : \text{nat}. \Lambda \alpha. \lambda f : \alpha \rightarrow \alpha. \lambda x : \alpha. n f (m f x)$.

- By iterating the addition, it is very easy to find a proof of $\forall xy : \text{nat}, N x \multimap N y \multimap !N(\text{mult } xy)$. Alas in order to build the scheme of bounded product in the following, we will need to find a proof of $\forall xy : \text{nat}, N x \multimap N y \multimap N(\text{mult } xy)$. The proof has been found and checked using a proof assistant based on our system, but it is too big to fit in there. The λ -term extracted from this proof is $\lambda nm : \text{nat}. \Lambda \alpha. \lambda f : \alpha \rightarrow \alpha. n \alpha (m (\alpha \rightarrow \alpha) (\lambda g : \alpha \rightarrow \alpha. \lambda x : \alpha. f(g x))) (\lambda x : \alpha. x)$.
- We can implement the predecessor function by proving $\vdash \forall x, N x \multimap N(\text{pred } x)$. The proof is not so easy: you have to instantiate a second-order quantifier with $x \mapsto (X p(x) \multimap X x) \otimes X p(x)$. It corresponds to a very standard technique for implementing the predecessor of n in λ -calculus: we iterate the function $(a, b) \mapsto (a + 1, a)$ n times on $(0, 0)$ and then we use the second projection to retrieve $n - 1$.
- Then it is easy to implement the subtraction by proving $\vdash \forall xy, N x \multimap N y \multimap !N(\text{minus } xy)$ with the induction principle $N y$.

- The following proof is called coercion (in [2]), it will allow us to replace occurrences of Nx at a negative position by $!Nx$. Let H be the formula $\forall y, Ny \multimap N(sy)$.

$$\begin{array}{c}
\frac{\frac{N0 \multimap Nx \vdash N0 \multimap Nx}{N0 \multimap Nx \vdash Nx} \quad \frac{\text{proof for zero}}{\vdash N0}}{\vdash!(N0 \multimap Nx) \multimap !Nx} \quad \frac{\frac{Nx \vdash Nx}{Nx \vdash !H \multimap !(N0 \multimap Nx)} \quad \frac{\text{proof for successor}}{\vdash !H}}{Nx \vdash !(N0 \multimap Nx)} \\
\hline
\frac{Nx \vdash !Nx}{\vdash \forall x, Nx \multimap !Nx}
\end{array}$$

Using this we can now bring every proof of totality

$$\vdash \forall x_1, \dots, x_n, !^{k_1}Nx_1 \multimap \dots \multimap !^{k_n}Nx_n \multimap !^k N(f x_1 \dots x_n)$$

to a “normal form”

$$\vdash \forall x_1, \dots, x_n, Nx_1 \multimap \dots \multimap Nx_n \multimap !^k N(f x_1 \dots x_n).$$

- The composition scheme is implemented by the following proof (where $s = \sum_{i=1}^q k_i$ and where $A^{(p)}$ means A is duplicated p times).

$$\begin{array}{c}
\frac{\frac{\text{proof for } g_1}{Nx_1, \dots, Nx_q \vdash !^{k_1} N(g_1 x_1 \dots x_q)} \quad \dots \quad \frac{\text{proof for } g_p}{Nx_1, \dots, Nx_q \vdash !^{k_p} N(g_p x_1 \dots x_q)} \quad \pi}{\frac{(Nx_1)^{(p)}, \dots, (Nx_q)^{(p)} \vdash !^{s+k} N(f(g_1 x_1 \dots x_q) \dots (g_p x_1 \dots x_q))}{(!Nx_1)^{(p)}, \dots, (!Nx_q)^{(p)} \vdash !^{s+k+1} N(f(g_1 x_1 \dots x_q) \dots (g_p x_1 \dots x_q))} \\
\hline
\frac{!Nx_1, \dots, !Nx_q \vdash !^{s+k+1} N(f(g_1 x_1 \dots x_q) \dots (g_p x_1 \dots x_q))}{Nx_1, \dots, Nx_q \vdash !^{s+k+1} N(f(g_1 x_1 \dots x_q) \dots (g_p x_1 \dots x_q))} \\
\hline
\vdash \forall x_1 \dots x_n, Nx_1 \multimap \dots \multimap Nx_q \multimap !^{s+k+1} N(f(g_1 x_1 \dots x_q) \dots (g_p x_1 \dots x_q)) \\
\\
\frac{\text{proof for } f}{\frac{N(g_1 x_1 \dots x_q), \dots, N(g_p x_1 \dots x_q) \vdash !^k N(f(g_1 x_1 \dots x_q) \dots (g_p x_1 \dots x_q))}{!^s N(g_1 x_1 \dots x_q), \dots, !^s N(g_p x_1 \dots x_q) \vdash !^{s+k} N(f(g_1 x_1 \dots x_q) \dots (g_p x_1 \dots x_q))} \\
\hline
!^s N(g_1 x_1 \dots x_q), \dots, !^s N(g_p x_1 \dots x_q) \vdash !^{s+k} N(f(g_1 x_1 \dots x_q) \dots (g_p x_1 \dots x_q)) \\
\hline
\vdash !^{k_1} N(g_1 x_1 \dots x_q) \multimap \dots \multimap !^{k_p} N(g_p x_1 \dots x_q) \multimap !^{s+k} N(f(g_1 x_1 \dots x_q) \dots (g_p x_1 \dots x_q)) \\
\pi
\end{array}$$

- Finally, the bounded sum is implemented by the following proof of $!(\forall y, Ny \multimap !^k N(fy)) \multimap \forall n, Nn \multimap !^{k+2} N(\text{sum } f n)$. The key idea in this proof is to use the induction principle of Nn with the predicate $x \mapsto Nx \otimes !^k N(\text{sum } f x)$. Let H be the formula $\forall y, Ny \multimap !^k N(fy)$ and K_1 be the formula

$$\forall y, !(Ny \otimes !^k N(\text{sum } f y)) \multimap !(N(sy) \otimes !^k N(\text{sum } f(sy)))$$

and K_2 the formula $!(N0 \otimes !^k N(\text{sum } f 0)) \multimap !(Nn \otimes !^k N(\text{sum } f n))$.

and we obtain the bounded product by replacing proofs for zeros by proof for ones and the proof for addition by a proof for multiplication.

- [1] Patrick Baillot (2004): *Type inference for light affine logic via constraints on words*. *Theoretical computer science* 328, pp. 289 – 323.
- [2] Vincent Danos & Jean-Baptiste Joinet (2001): *Linear Logic & Elementary Time*. *Information and Computation* 183.
- [3] Jean-Yves Girard (1998): *Light Linear Logic*. *Information and Computation* 143(2), pp. 175 – 204.
- [4] Jean-Louis Krivine (1993): *Lambda-calculus, types and models*. Ellis Horwood.
- [5] Marc Lasson (2009): *A pure type system for second-order arithmetic*. Not yet published .
- [6] Simone Martini & Paolo Coppola (2001): *Typing Lambda Terms in Elementary Logic with Linear Constraints*. *Typed Lambda Calculi and Applications* , pp. 76 – 90.
- [7] H.E. Rose (1984): *Sub-recursion: functions and hierarchy*. Oxford University Press.